**Exercise 3: Sorting Customer Orders**

Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).

Bubble Sort

- Works by repeatedly iterating through the data and swapping adjacent elements if they are in the wrong order.

- Continues iterating until no more swaps are needed, indicating that the data is sorted.

- Time complexity: O(n^2)

- Space complexity: O(1)

- Simple to implement, but not efficient for large datasets.

Insertion Sort

- Works by iterating through the data one element at a time, inserting each element into its proper position in the sorted portion of the data.

- Time complexity: O(n^2)

- Space complexity: O(1)

- Efficient for small datasets or nearly sorted data.

Quick Sort

- Works by selecting a pivot element, partitioning the data around the pivot, and recursively sorting the subarrays.

- Time complexity: O(n log n) on average, O(n^2) in the worst case.

- Space complexity: O(log n)

- Efficient and widely used, but can be slow for already sorted or nearly sorted data.

Merge Sort

- Works by dividing the data into smaller chunks, sorting each chunk, and then merging the sorted chunks back together.

- Time complexity: O(n log n)

- Space complexity: O(n)

- Stable and efficient, but requires extra memory for the merge process.

These are just some of the most common sorting algorithms, and there are many others, each with their own strengths and weaknesses. The choice of algorithm depends on the specific use case and requirements.

//Java

public class Order {

private int orderId;

private String customerName;

private double totalPrice;

public Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

// Getters and setters

}

public class BubbleSort {

public static void sortOrders(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {

// Swap orders

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

}

public class QuickSort {

public static void sortOrders(Order[] orders) {

quickSort(orders, 0, orders.length - 1);

}

private static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].getTotalPrice();

int i = (low - 1);

for (int j = low; j < high; j++) {

if (orders[j].getTotalPrice() < pivot) {

i++;

// Swap orders

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

// Swap orders

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

}

Analysis

Compare the performance (time complexity) of Bubble Sort and Quick Sort.

- Bubble Sort:

- Best-case scenario: O(n)

- Average-case scenario: O(n^2)

- Worst-case scenario: O(n^2)

- Quick Sort:

- Best-case scenario: O(n log n)

- Average-case scenario: O(n log n)

- Worst-case scenario: O(n^2)

Discuss why Quick Sort is generally preferred over Bubble Sort.

Quick Sort is generally preferred over Bubble Sort because of its better average-case and worst-case time complexity (O(n log n) vs O(n^2)). This makes Quick Sort more efficient for large datasets. However, Bubble Sort can be suitable for small datasets or nearly sorted data.